Programming in C/C++
Exercises set five: grammatical parsers

Christiaan Steenkist
Jaime Betancor Valado
Remco Bos

March 1, 2017

Exercise 30, calculator

We got a calculator grammar.

Code listings

Listing 1: grammar.gr

```
%scanner "Scanner.h"

%token NUMBER

%left ’+’ ’-’
%left ’*’ ’/’
%right ’$’

%%

input:
  input line
|
  // empty
;

line:
  ’\n’
```

```
|
  expr '\n'
|
  error '\n'
;

expr:
  NUMBER
|
  expr '+' expr
|
  expr '-' expr
|
  expr '*' expr
|
  expr '/' expr
|
  '-' expr %prec '$'
|
  '$' expr
|
  '(' expr ')'
;
```

Exercise 31, conflicts

We fixed numerous conflicts.

First grammar

The first revision of the grammar solves the reduce/reduce conflicts that are caused by the `NUMBER` branches in the `expr` and `number` nonterminals. To solve this we removed the `expr` case because it is already represented in `number`.

```
%token NUMBER

%%

expr:
    number
|
    expr '+' expr
|
    expr '-' expr
|
  // empty
;

number:
    NUMBER
;
```

Second grammar

In this grammar we also solved the shift/reduce conflicts which were caused because '+' and '-' have equal priority and no explicitly specified resolution for these types of conflicts. We set them to equal priority but reduce and not shift as is customary with these operators. Also to avoid very weird cases we modified the grammar a little.

```
%token NUMBER

%left '+' '-'

%%
expr:
  exprpart
|
  // empty
;

exprpart:
    number
|
    exprpart '+' exprpart
|
    exprpart '-' exprpart
;

number:
    NUMBER
```

```
;
```

Exercise 32, priorities

We added a new priority to a grammar.

Code listings

Listing 2: grammar.gr

```
%token NR

%left NOTEQUAL
%left '+'
%left '*'
%left '^'
%right '-'

%%

expr:
  NR
|
  '-' expr
|
  expr '+' expr
|
  expr '*' expr
|
  expr NOTEQUAL expr
|
  expr '^' expr
;
```

Exercise 35, separated lists

We fixed the list grammar so it does what we want it to, the right way this time.

Design

There are three kinds of list, plain, separated and empty. The types plain and separated have non-terminals but the empty list does not as this is not necessary. Each list can only have one token type in it which consists of `plain_(token)_seg` or `sep_(token)_seg` non-terminals that we made for each token. Tokens are reduced into the proper list `seg`ment at the start of the list. A program could maintain this structure in the grammar file, adding or removing rules for certain tokens.

Code listings

Listing 3: grammar.gr

```
%token WORD
%token INT
%token FLOAT

%scanner Scanner.h

%%

list:
  plain
|
  separated
|
  // empty
;

// Plain list

plain:
  plain_word_seg
|
  plain_int_seg
|
  plain_float_seg
;

plain_word_seg:
  plain_word_seg WORD
```

```
|
  WORD
;

plain_int_seg:
  plain_int_seg INT
|
  INT
;

plain_float_seg:
  plain_float_seg FLOAT
|
  FLOAT
;

// Separated list
separated:
  sep_word_seg WORD
|
  sep_int_seg INT
|
  sep_float_seg FLOAT
;

sep_word_seg:
  sep_word_seg WORD ','
|
  WORD ','
;

sep_int_seg:
  sep_int_seg INT ','
|
  INT ','
;

sep_float_seg:
  sep_float_seg FLOAT ','
|
    FLOAT ','
;
```